Heap Binaria

Definición

Una cola de prioridad es una estructura de datos que permite al menos dos operaciones:

- Insert

  Inserta un elemento en la estructura

- DeleteMin

  Encuentra, recupera y elimina el elemento mínimo

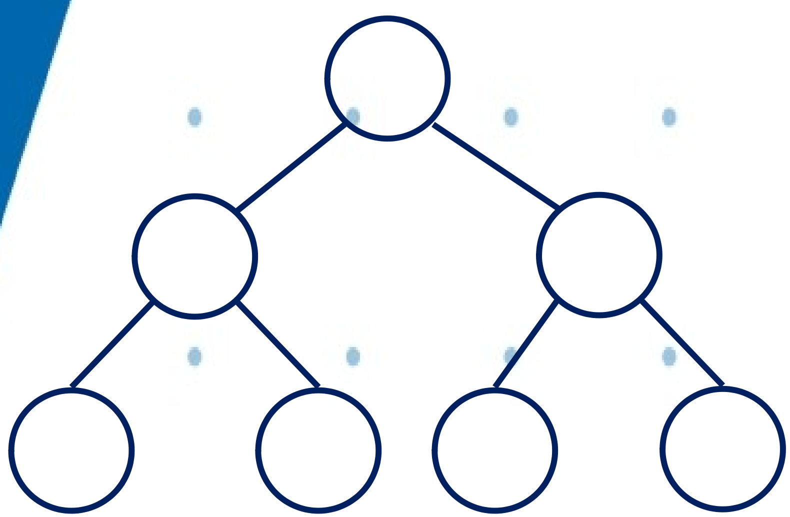
Heap Binaria

- Es una implementación de colas de prioridad que no usa punteros y permite implementar ambas operaciones con O(log N) operaciones en el peor caso
- Cumple con dos propiedades:
  - ✓ Propiedad estructural
  - ✓ Propiedad de orden

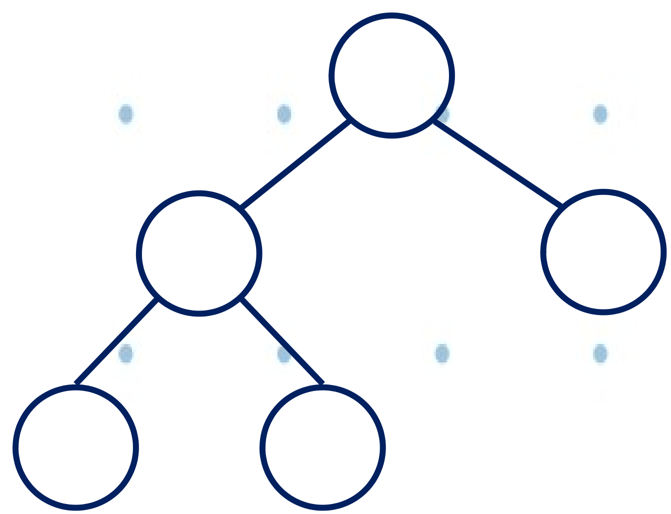
Propiedad estructural

Una heap es un árbol binario completo

Árbol binario lleno



Árbol binario completo

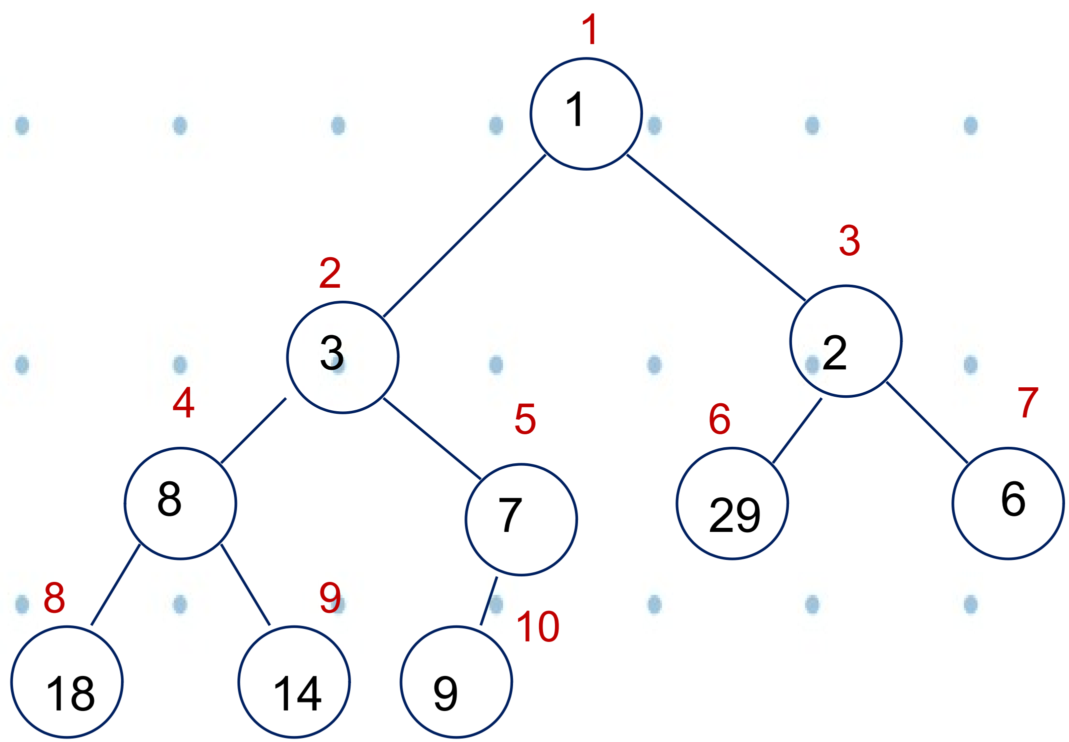


Propiedad estructural (cont.)

- Dado que un árbol binario completo es una estructura de datos regular, puede almacenarse en un arreglo, tal que:
  - La raíz está almacenada en la posición 1
  - Para un elemento que está en la posición i:
    - El hijo izquierdo está en la posición 2*i
    - El hijo derecho está en la posición 2*i + 1
    - El padre está en la posición i/2

Propiedad estructural (cont.)

El árbol que vimos como ejemplo, puede almacenarse de la siguiente manera:



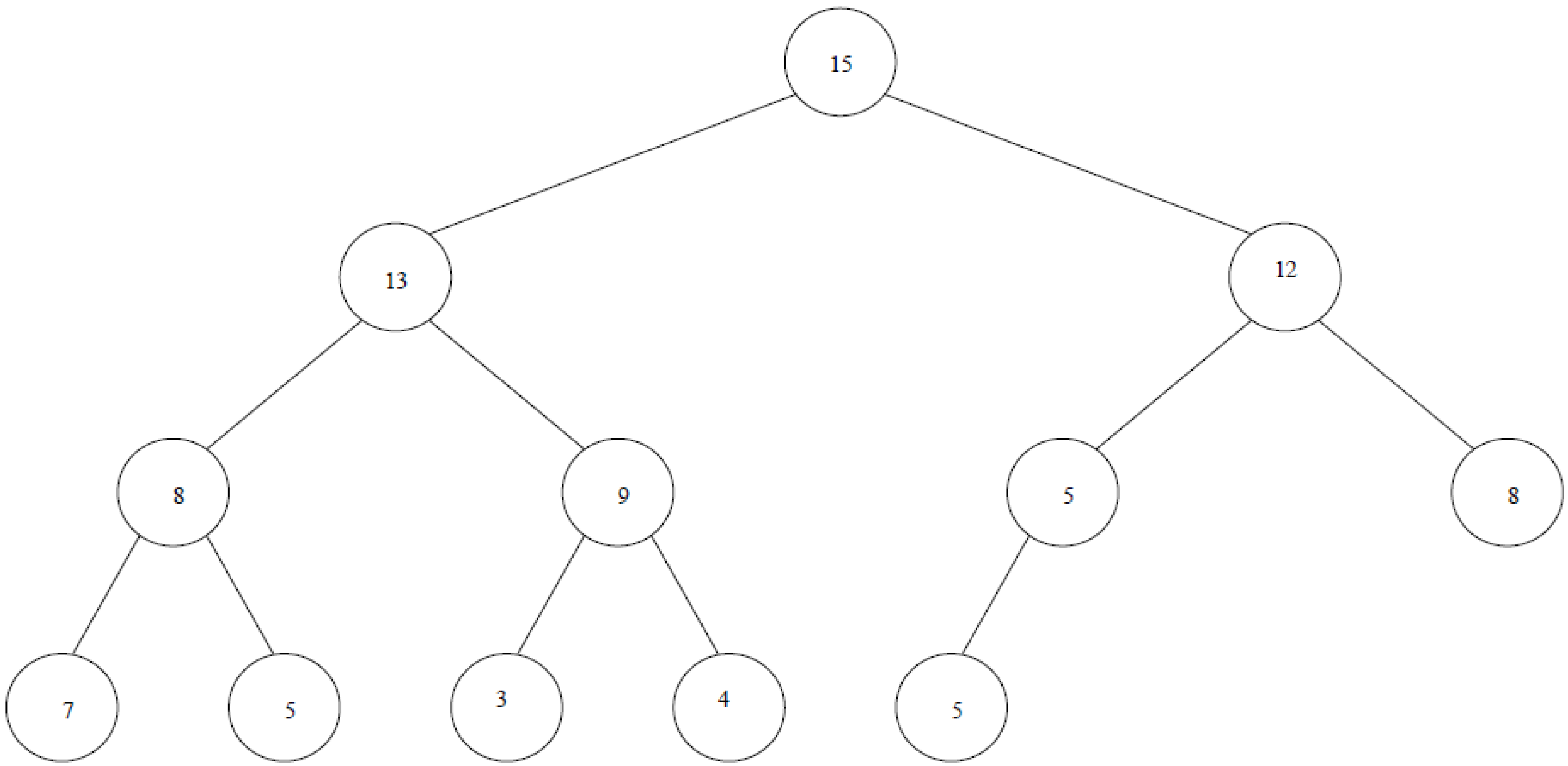
1	3	2	8	7	29	6	18	14	9
1	2	3	4	5	6	7	8	9	10

Propiedad de orden

- MinHeap
  - El elemento mínimo está almacenado en la raíz
  - El dato almacenado en cada nodo es menor o igual al de sus hijos
- MaxHeap
  - Se usa la propiedad inversa

Propiedad de orden (cont.)

Ejemplo de MaxHeap:



15	13	12	8	9	5	8	7	5	3	4	5
1	2	3	4	5	6	7	8	9	10	11	12

Implementación de Heap

Una heap H consta de:

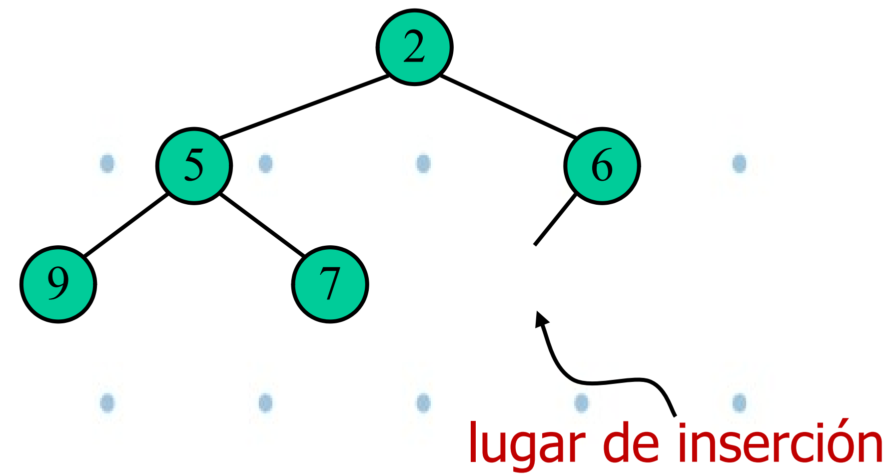
- *Un arreglo que contiene los datos*
- *Un valor que me indica el número de elementos almacenados*

Ventaja:

- No se necesita usar punteros
- Fácil implementación de las operaciones

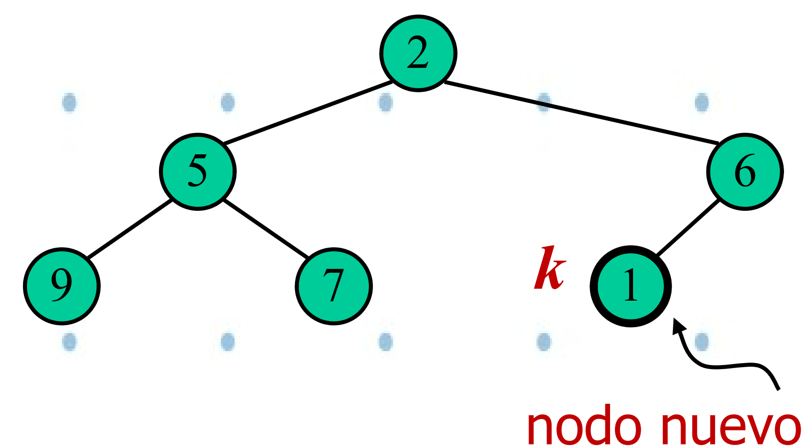
Operación: Insert

- El dato se inserta como último ítem en la heap
  - La propiedad de la heap puede ser violada



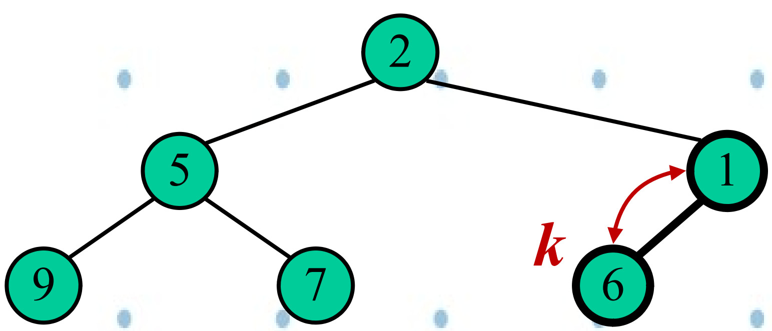
- Se debe hacer un filtrado hacia arriba para restaurar la propiedad de orden

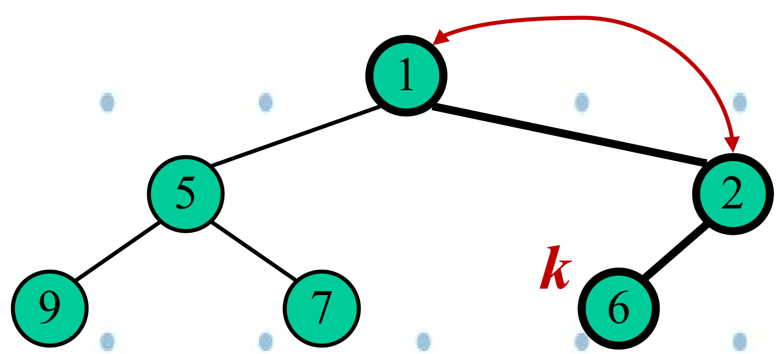
Inserto el 1



Insert: Filtrado hacia arriba

- El filtrado hacia arriba restaura la propiedad de orden intercambiando ***k*** a lo largo del camino hacia arriba desde el lugar de inserción
- El filtrado termina cuando la clave ***k*** alcanza la raíz o un nodo cuyo padre tiene una clave menor
- Ya que el algoritmo recorre la altura de la heap, tiene *O(log n)* intercambios



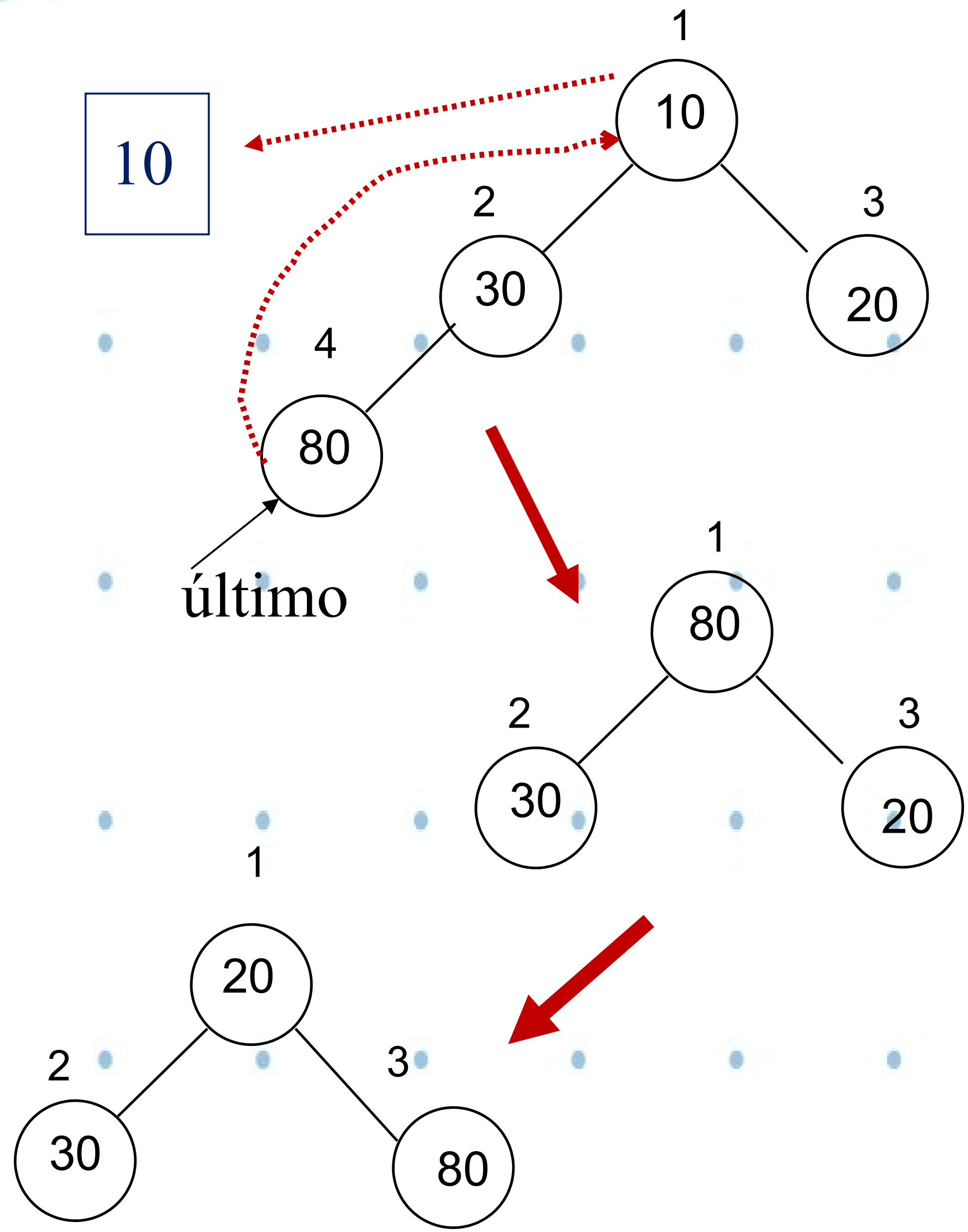


Operación: DeleteMin

- Guardo el dato de la raíz

- Elimino el último elemento y lo almaceno en la raíz

- Se debe hacer un filtrado hacia abajo para restaurar la propiedad de orden



DeleteMin: Filtrado hacia abajo

- Es similar al filtrado hacia arriba
- El filtrado hacia abajo restaura la propiedad de orden intercambiando el dato de la raíz hacia abajo a lo largo del camino que contiene los hijos mínimos
- El filtrado termina cuando se encuentra el lugar correcto dónde insertarlo
- Ya que el algoritmo recorre la altura de la heap, tiene *O(log n)* operaciones de intercambio.

¿Cómo construir una heap a partir de una lista de elementos?

Para construir una heap a partir de una lista de *n* elementos:

✓ Se pueden insertar los elementos de a uno

⇒ se realizan (n log n) operaciones en total

✓ Se puede usar un algoritmo de orden lineal, es decir, proporcional a los n elementos ⇒ **BuildHeap**

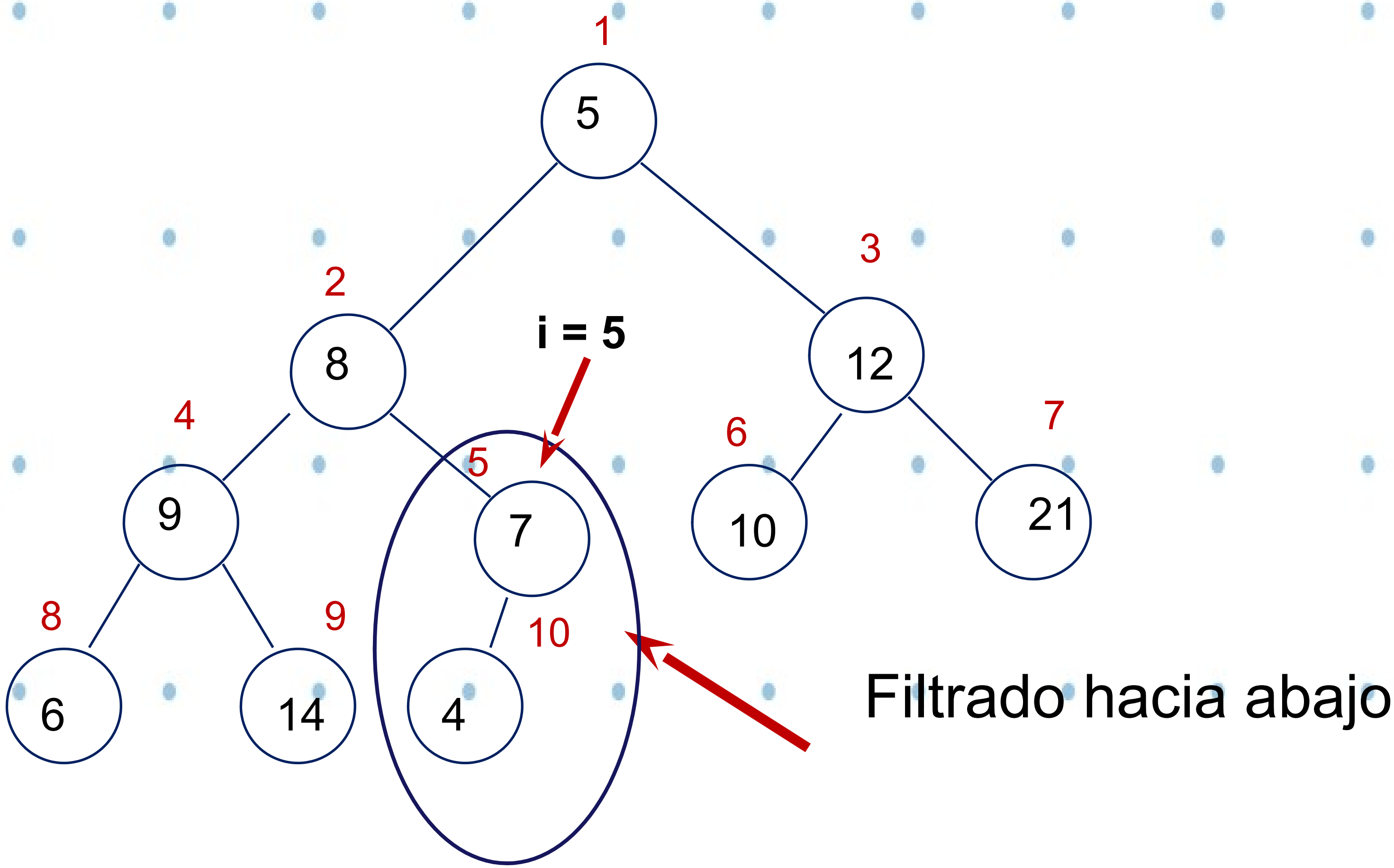
- Insertar los elementos desordenados en un árbol binario completo
- Filtrar hacia abajo cada uno de elementos

Algoritmo BuildHeap

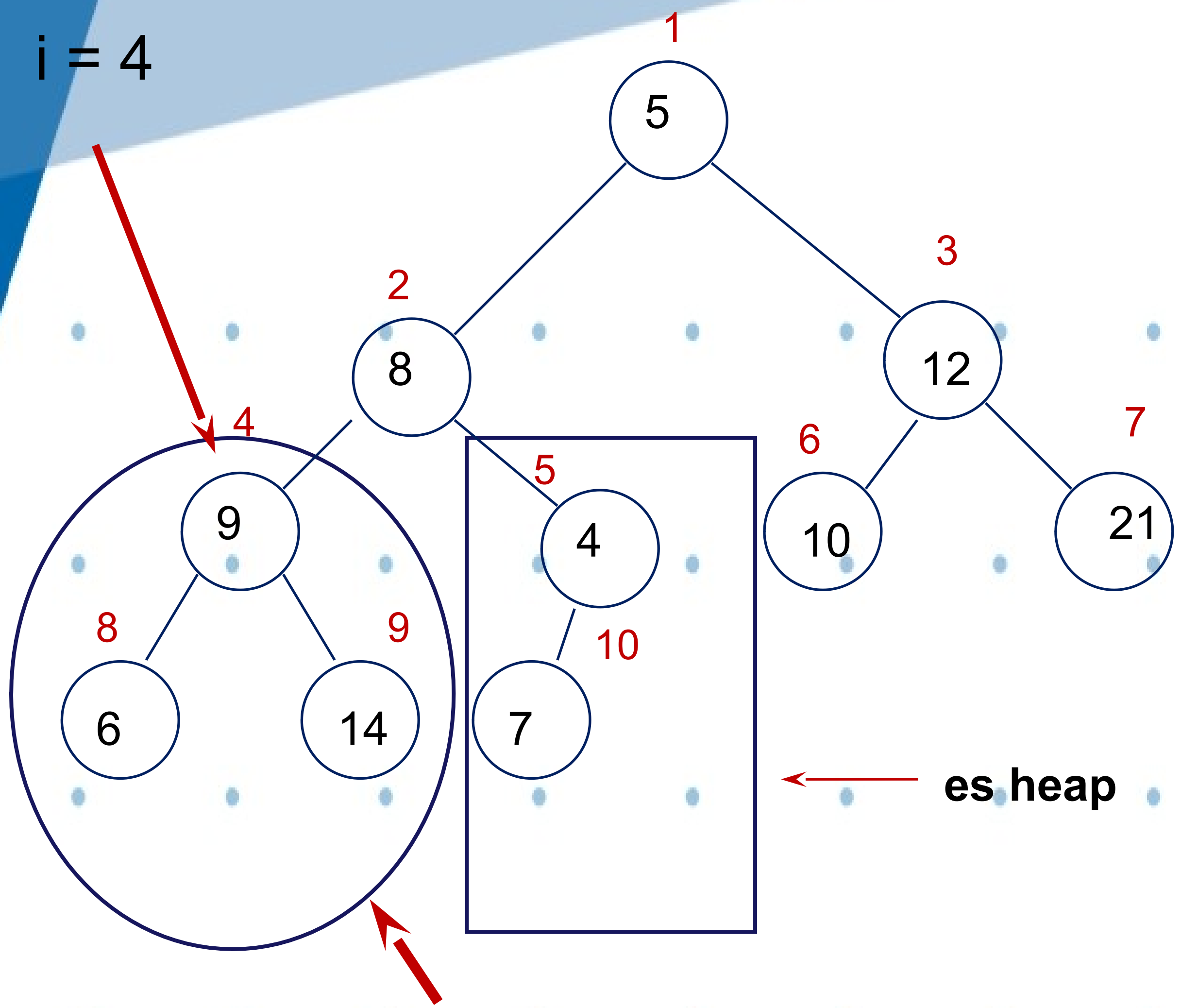
- Para filtrar:
  - se elige el menor de los hijos
  - se compara el menor de los hijos con el padre
- Se empieza filtrando desde el elemento que está en la posición (tamaño/2):
  - se filtran los nodos que tienen hijos
  - el resto de los nodos son hojas

BuildHeap

5	8	12	9	7	10	21	6	14	4
1	2	3	4	5	6	7	8	9	10

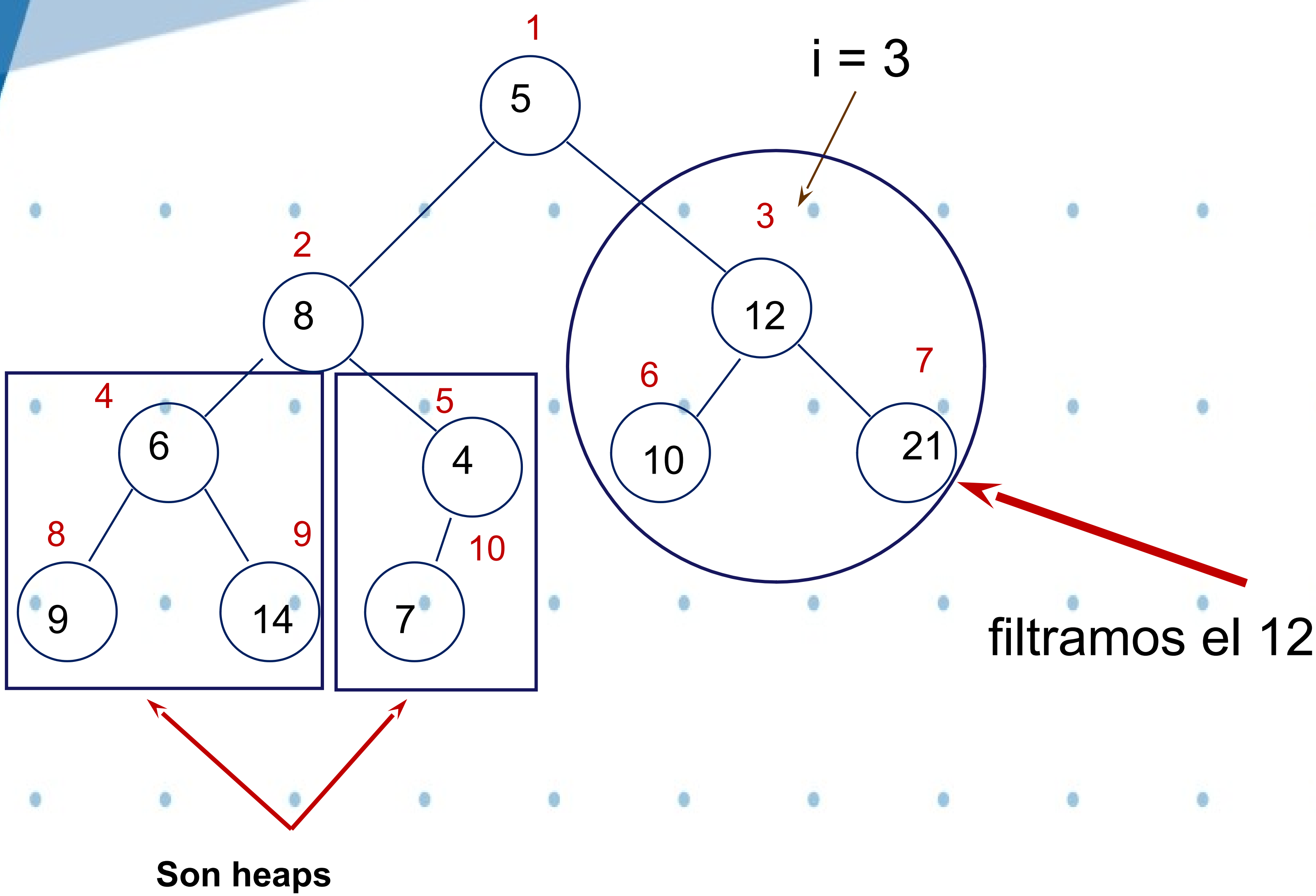


BuildHeap

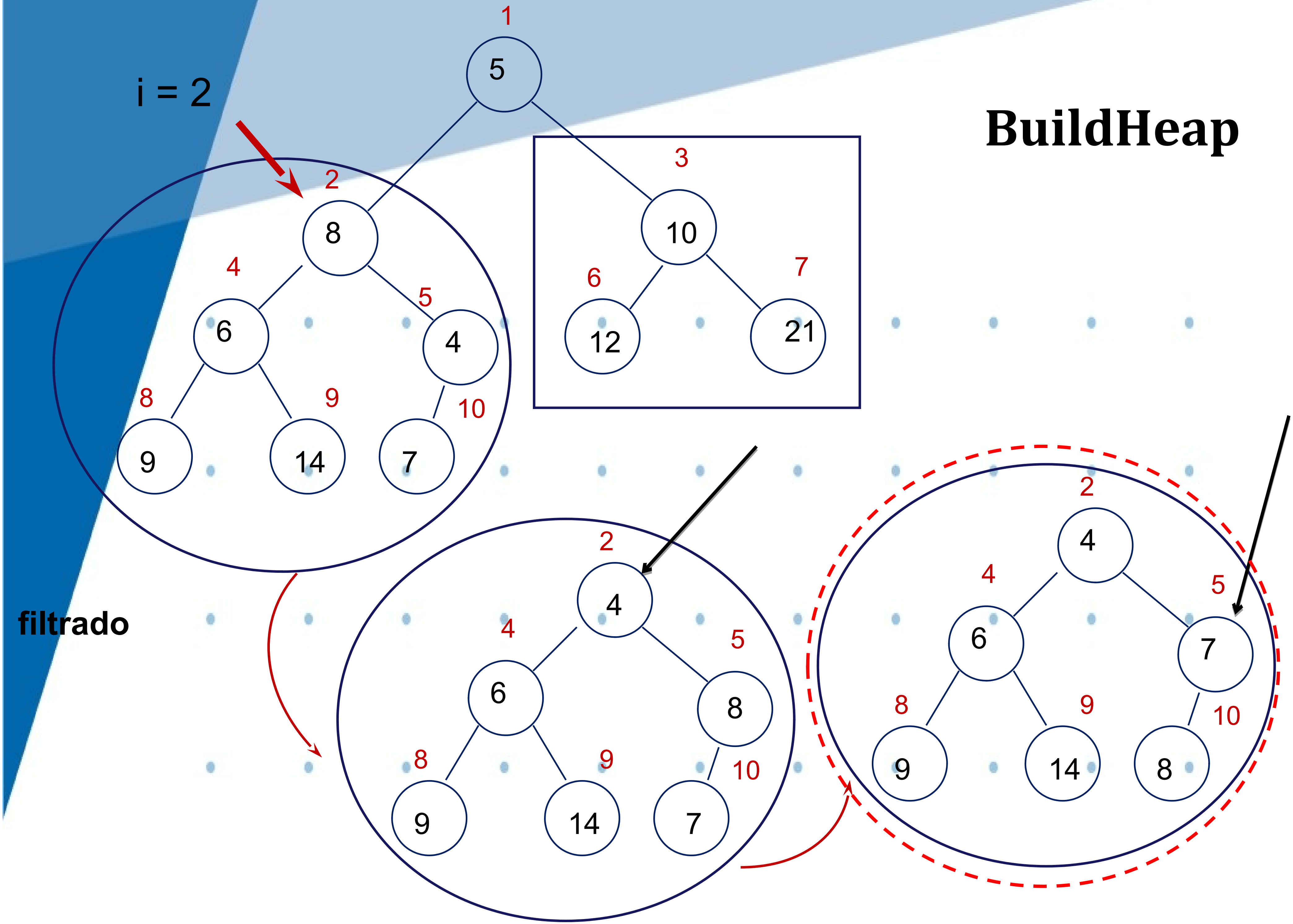


filtramos el 9

BuildHeap



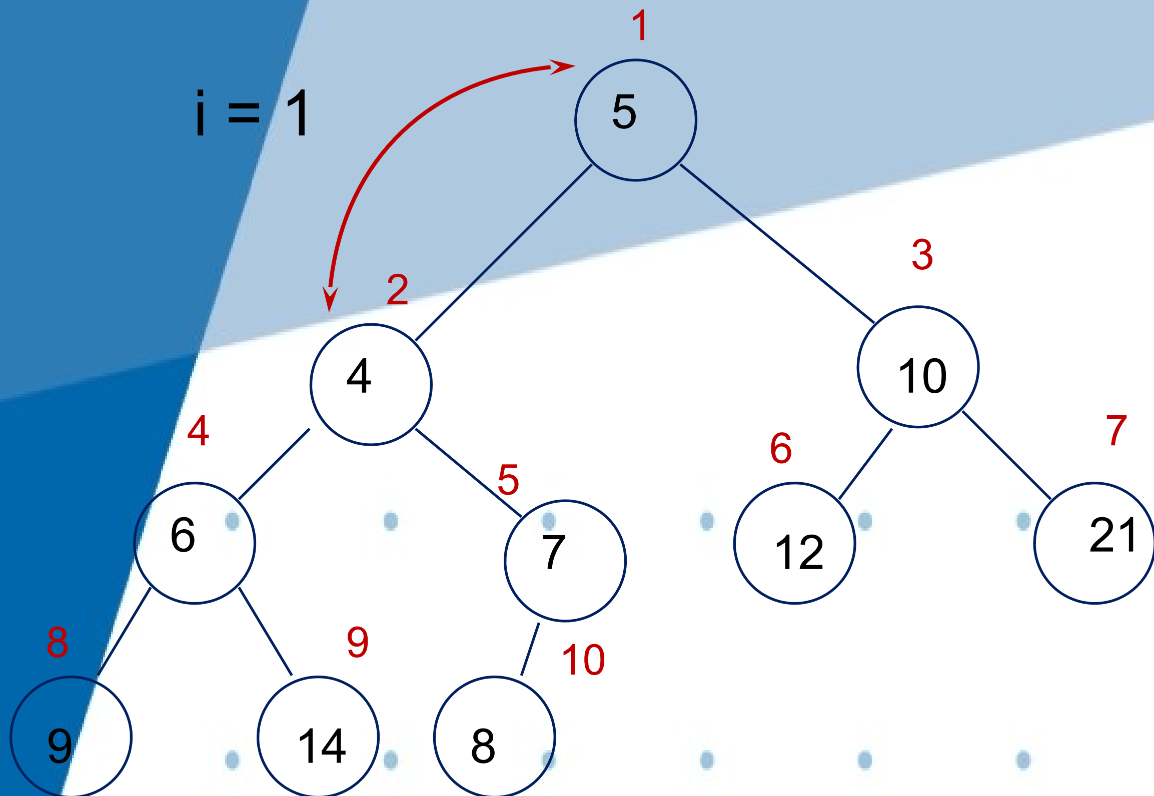
BuildHeap

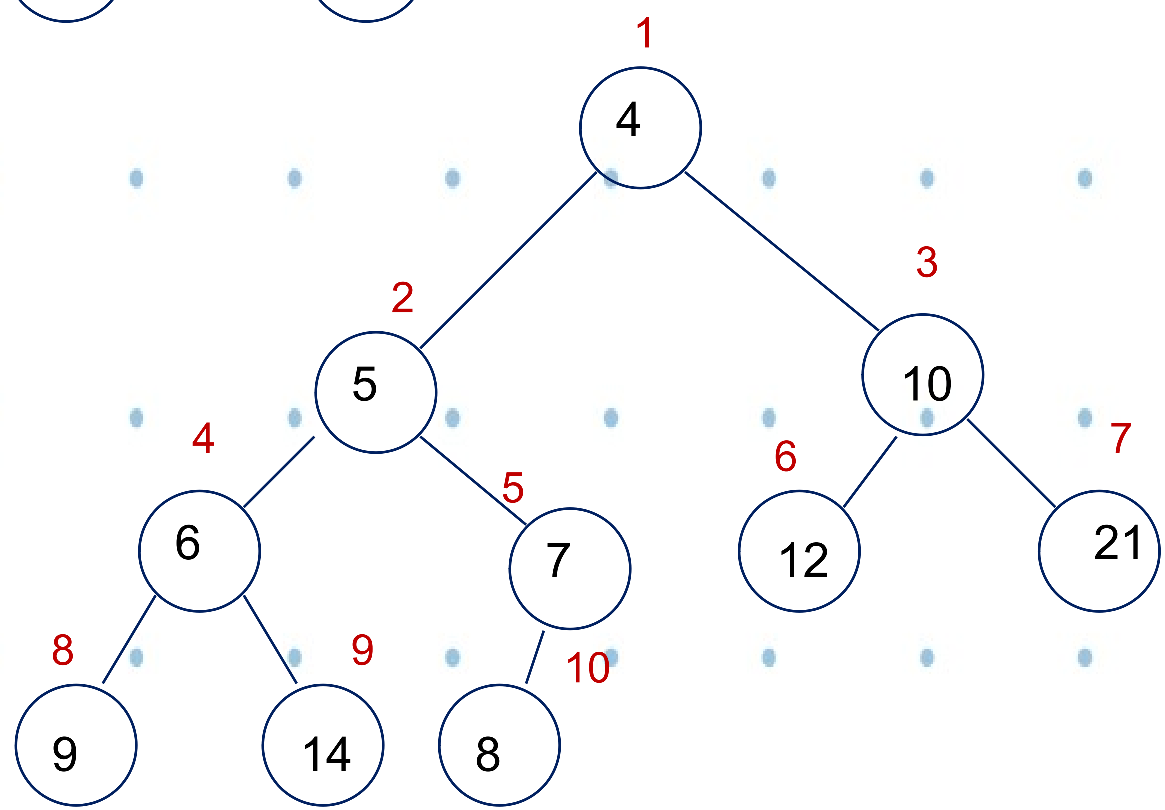


i = 2
filtrado
1
5
2
8
3
10
4
6
5
4
6
12
7
21
8
9
9
14
10
7
2
4
4
6
5
8
8
9
9
14
10
7
2
4
4
6
5
7
8
9
9
14
10
8

BuildHeap

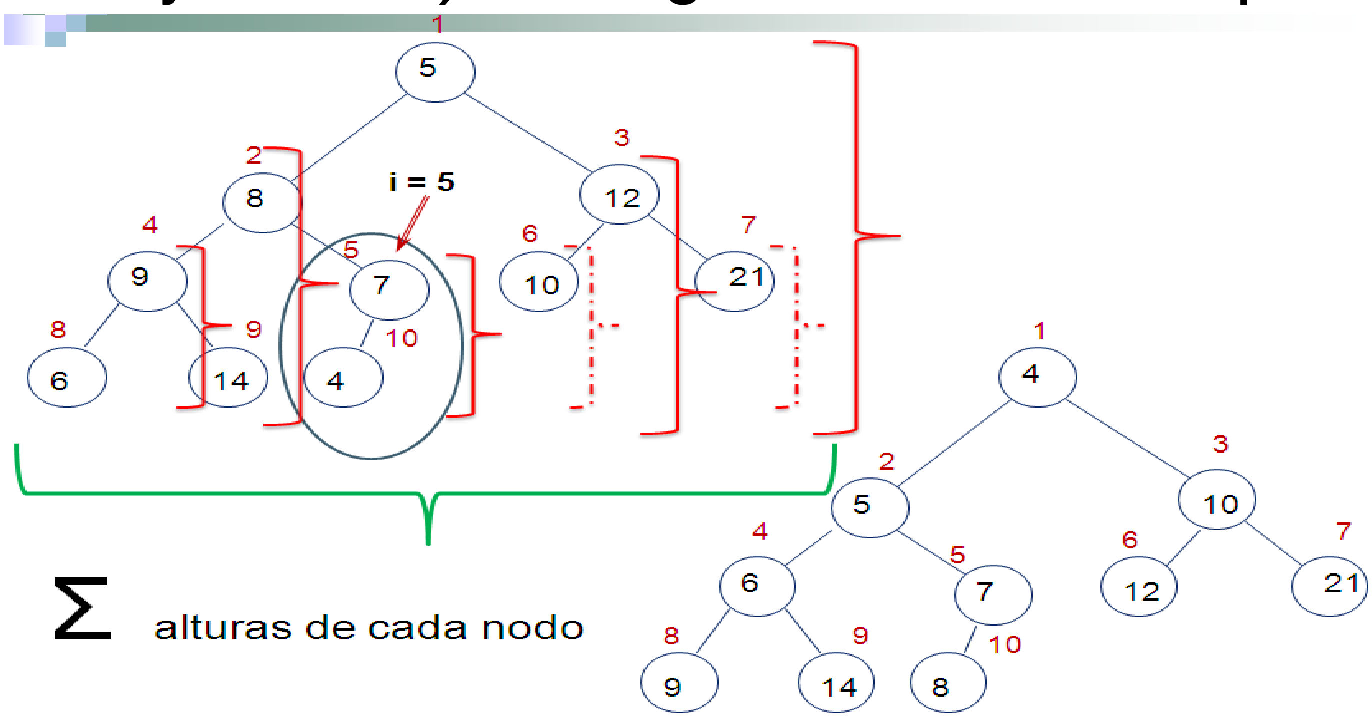
i = 1





BuildHeap

- Cantidad de Operaciones:
  - En el filtrado de cada nodo recorremos su altura.
  - Para acotar la cantidad de operaciones (tiempo de ejecución) del algoritmo BuildHeap, debemos



os

BuildHeap

Teorema:

En un árbol binario lleno de altura h que contiene $2^{h+1} - 1$ nodos, la suma de las alturas de los nodos es: $2^{h+1} - 1 - (h + 1)$

Demostración:

Un árbol tiene 2^i nodos de altura $h - i$

$$S = \sum_{i=0}^{h} 2^i (h-i)$$

$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots\ldots\ldots 2^{h-1}(1)$$

BuildHeap

$S = h + 2\,(h-1) + 4\,(h-2) + 8\,(h-3) + \ldots\ldots\ldots\ldots\ldots + 2^{h-1}\,(1) \quad (A)$

$-\quad -\quad -\quad -\quad -$

$2S = 2h + 4\,(h-1) + 8\,(h-2) + 16\,(h-3) + \ldots\ldots\ldots\ldots + 2^{h}\,(1) \quad (B)$

Restando las dos igualdades (B) – (A)

$S = -h + 2\,(h-(h-1)) + 4((h-1)-(h-2)) + 8\,((h-2)-(h-3)) + \ldots + 2^{h-1}(2-1) + 2^{h}$

$S = -h + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$

$S + 1 = -h + 1 + 2 + 4 + 8 + 16 + \ldots\ldots\ldots + 2^{h-1} + 2^{h}$

$S + 1 = -h + (2^{h+1} - 1)$

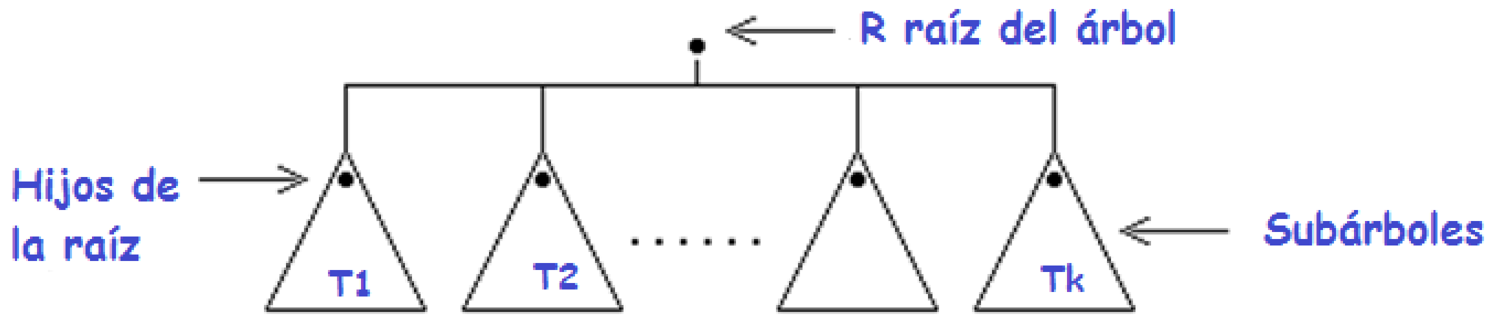
$S = (2^{h+1} - 1) - (h + 1)$

Árboles Generales

Definición

➢ *Un árbol es una colección de nodos, tal que:*

- *puede estar vacía. (Árbol vacío)*
- *puede estar formada por un nodo distinguido R, llamado **raíz** y un conjunto de árboles T_1, T_2, ...T_k, $k \geq 0$ (subárboles), donde la raíz de cada subárbol T_i está conectado a R por medio de una arista*



Descripción y terminología

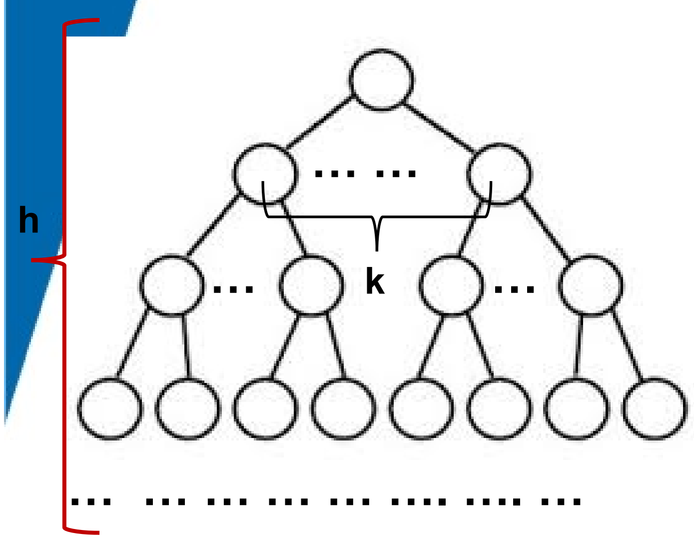
- ***Grado*** del árbol es el grado del nodo con mayor grado.
- ***Árbol lleno***: Dado un árbol T de grado ***k*** y altura ***h***, diremos que T es *lleno* si cada nodo interno tiene grado k y todas las hojas están en el mismo nivel (h).

Es decir, recursivamente, T es *lleno* si :
1.- T es un nodo simple (árbol lleno de altura 0), o
2.- T es de altura h y todos sus sub-árboles son llenos de altura h-1.

Descripción y terminología

- ***Árbol completo***: Dado un árbol T de grado ***k*** y altura ***h***, diremos que T es *completo* si es lleno de altura h-1 y el nivel h se completa de izquierda a derecha.
- ***Cantidad de nodos en un árbol <u>lleno</u>***:

  Sea T un árbol lleno de grado ***k*** y altura ***h***, la cantidad de nodos N es $\mathbf{(k^{h+1} - 1) / (k-1)}$ ya que:



Nivel 0 → k^0 nodos

Nivel 1 → k^1 nodos

Nivel 2 → k^2 nodos

Nivel 3 → k^3 nodos

……

$$\mathbf{N = k^0 + k^1 + k^2 + k^3 + \ldots + k^h}$$

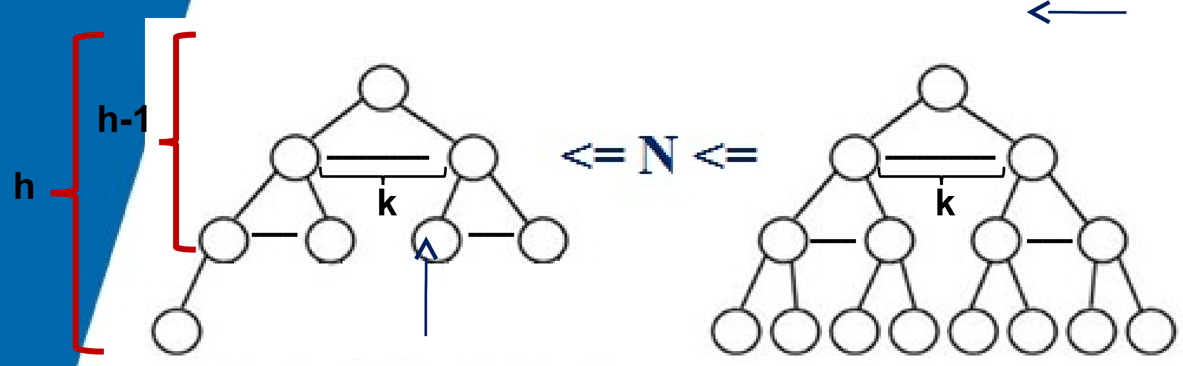
La suma de los términos de una serie geométrica de razón k es:

$\mathbf{(k^{h+1} - 1) / (k-1)}$

Descripción y terminología

- ***Cantidad de nodos en un árbol <u>completo</u>***:

Sea T un árbol completo de grado ***k*** y altura ***h***, la cantidad de nodos N varía entre $(k^{h} + k - 2) / (k-1)$ y $(k^{h+1} - 1) / (k-1)$ ya que …



- Si el árbol es lleno

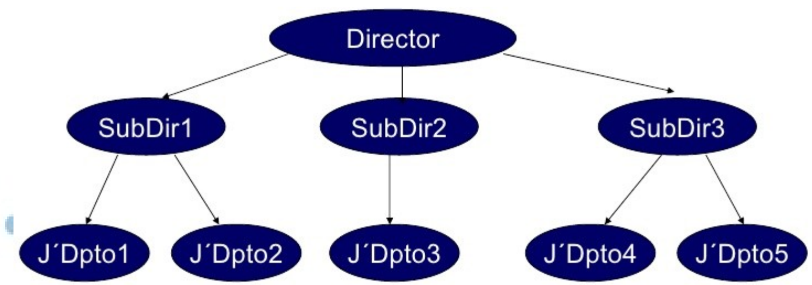
  $\mathbf{N} = (k^{h+1}-1)/(k-1)$

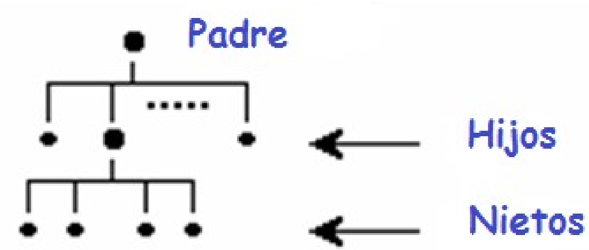
- Si no, el árbol es lleno en la altura *h-1* y tiene por lo menos un nodo en el nivel *h:*

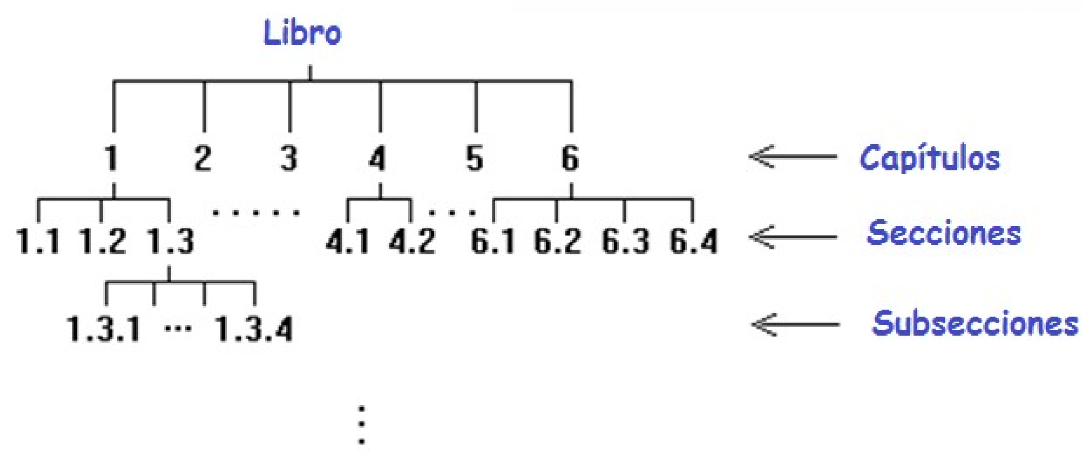
$\mathbf{N} = (k^{h-1+1}-1)/(k-1)+1=(k^{h} + k -2) / (k -1)$

Ejemplos

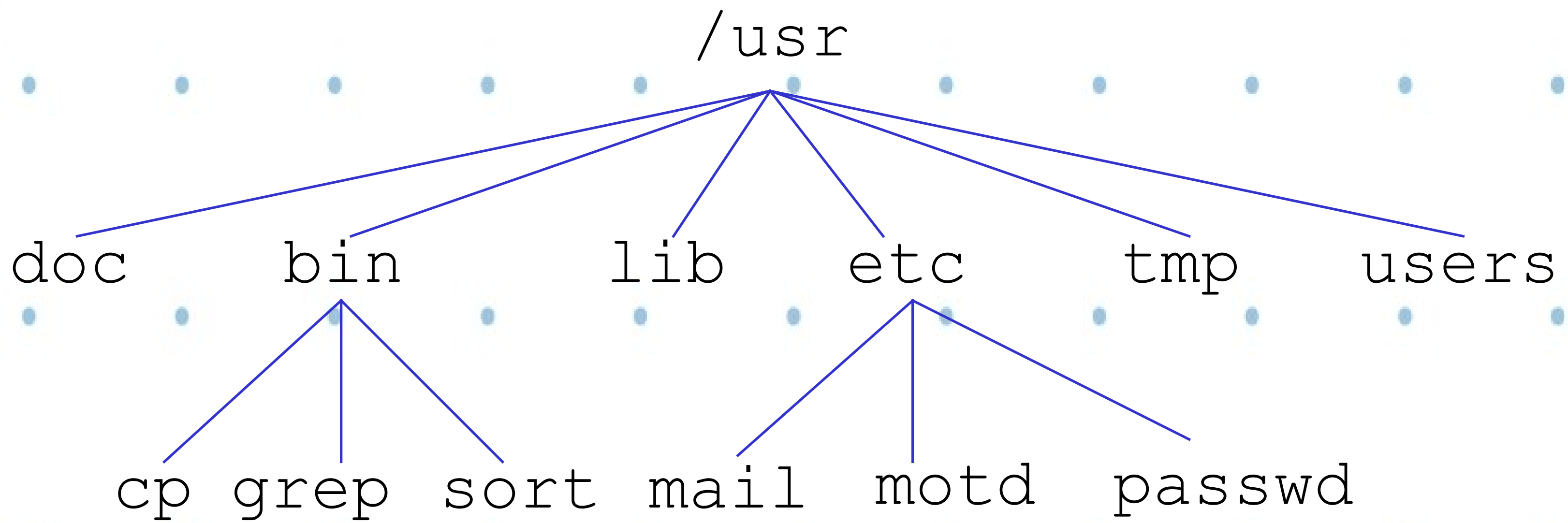
- Organigrama de una empresa
- Árboles genealógicos
- Taxonomía que clasifica organismos
- Sistemas de archivos
- Organización de un libro en capítulos y secciones







Ejemplo: Sistema de archivos



Representaciones

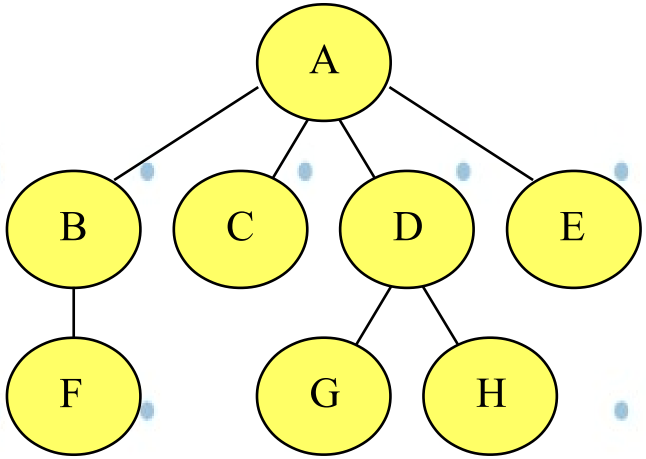
- Lista de hijos
  - Cada nodo tiene:
    - Información propia del nodo
    - Una lista de todos sus hijos
- Hijo más izquierdo y hermano derecho
  - Cada nodo tiene:
    - Información propia del nodo
    - Referencia al hijo más izquierdo
    - Referencia al hermano derecho

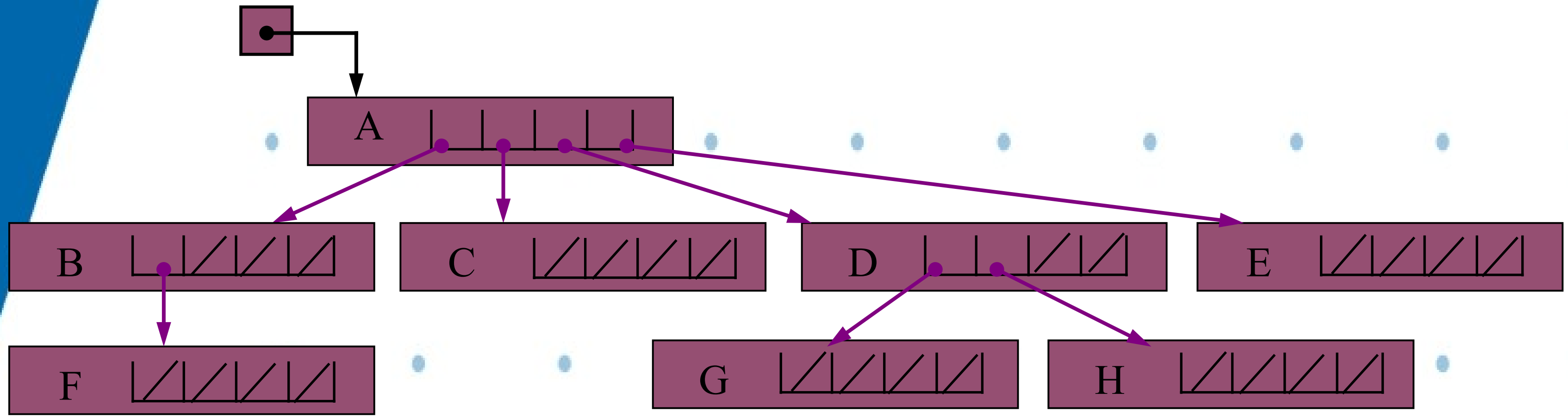
Representación: Lista de hijos

- La lista de hijos, puede estar implementada a través de:
  - Arreglos
    - Desventaja: espacio ocupado
  - Listas dinámicas
    - Mayor flexibilidad en el uso

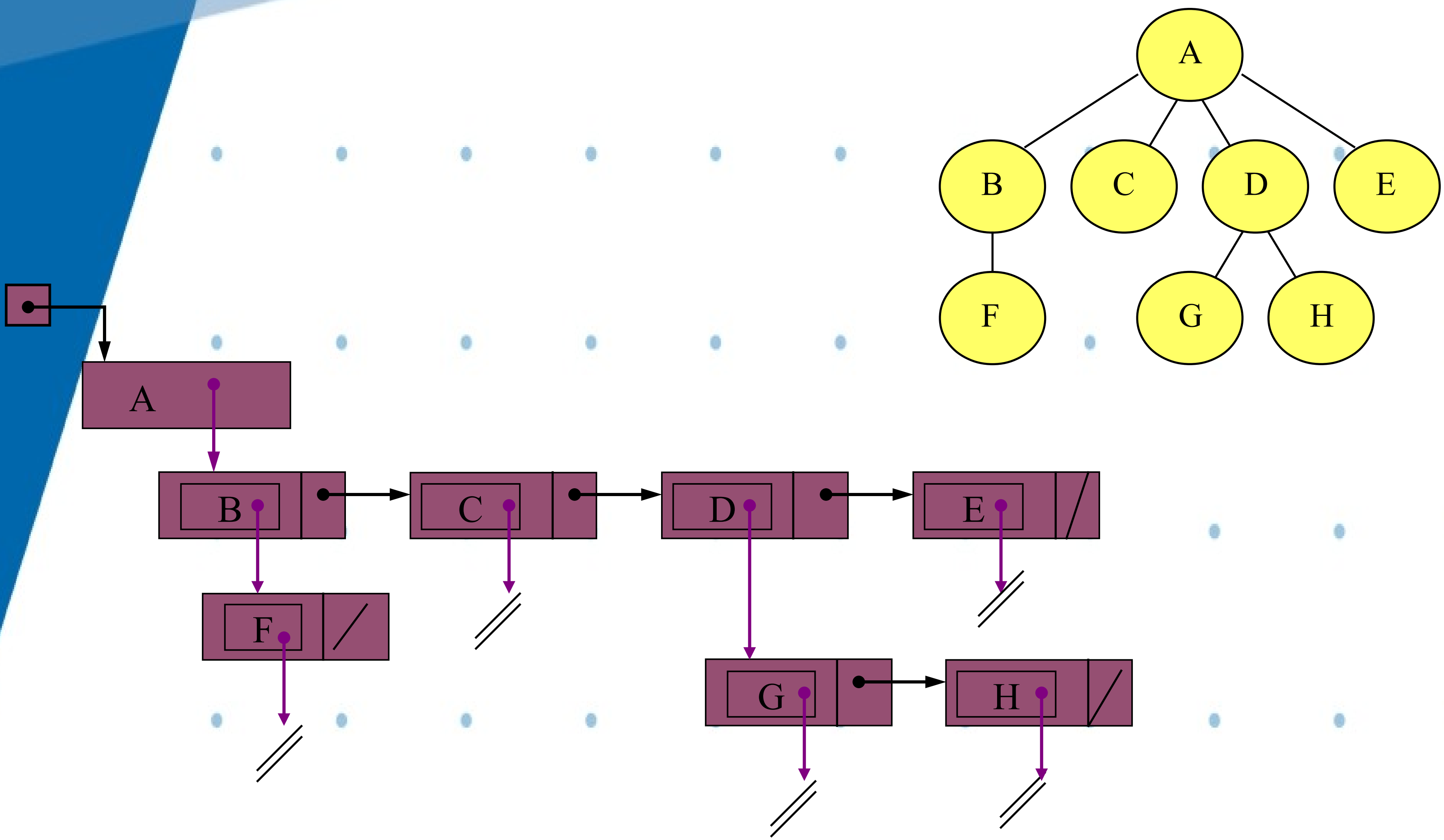
Representación: Lista de hijos

Implementada con Arreglos

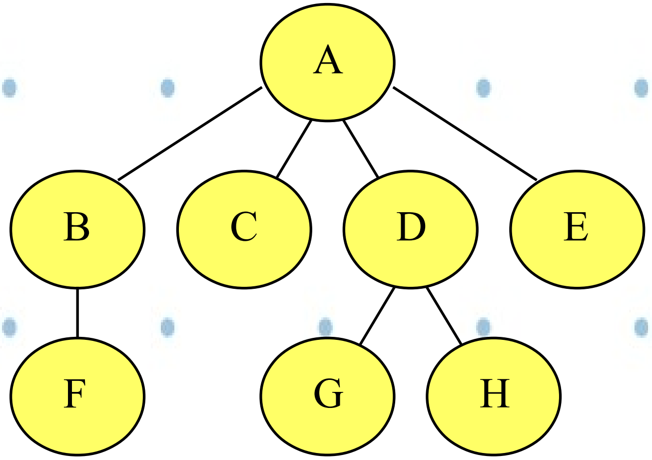


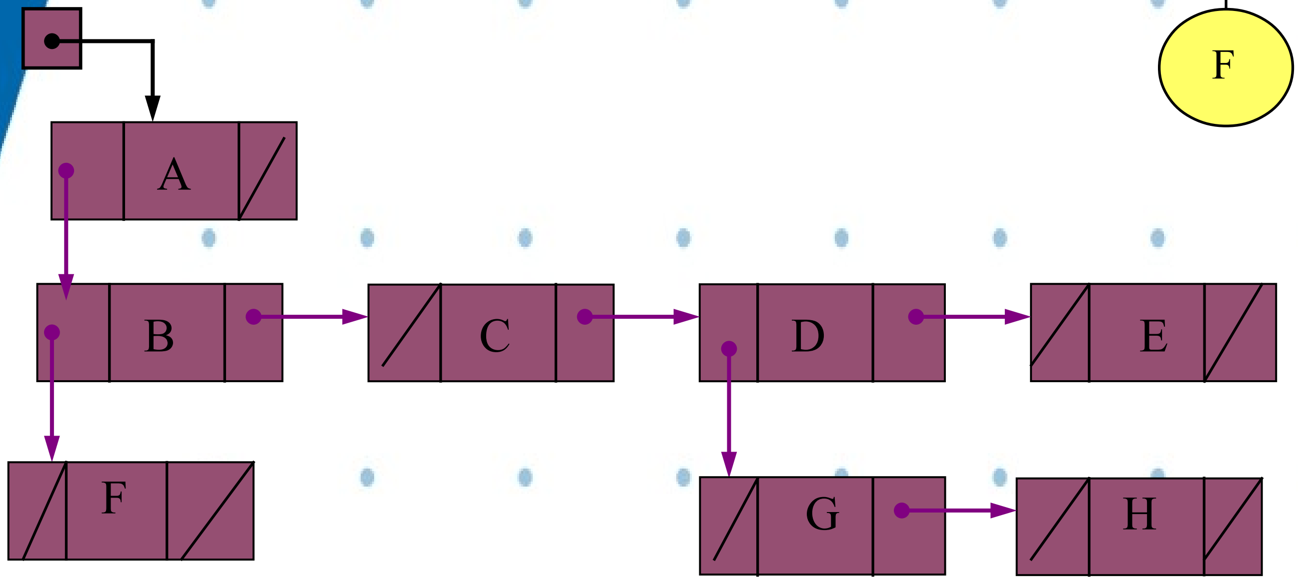


Representación: Lista de hijos
Implementada con Listas enlazadas



Representación: Hijo más izquierdo y hermano derecho

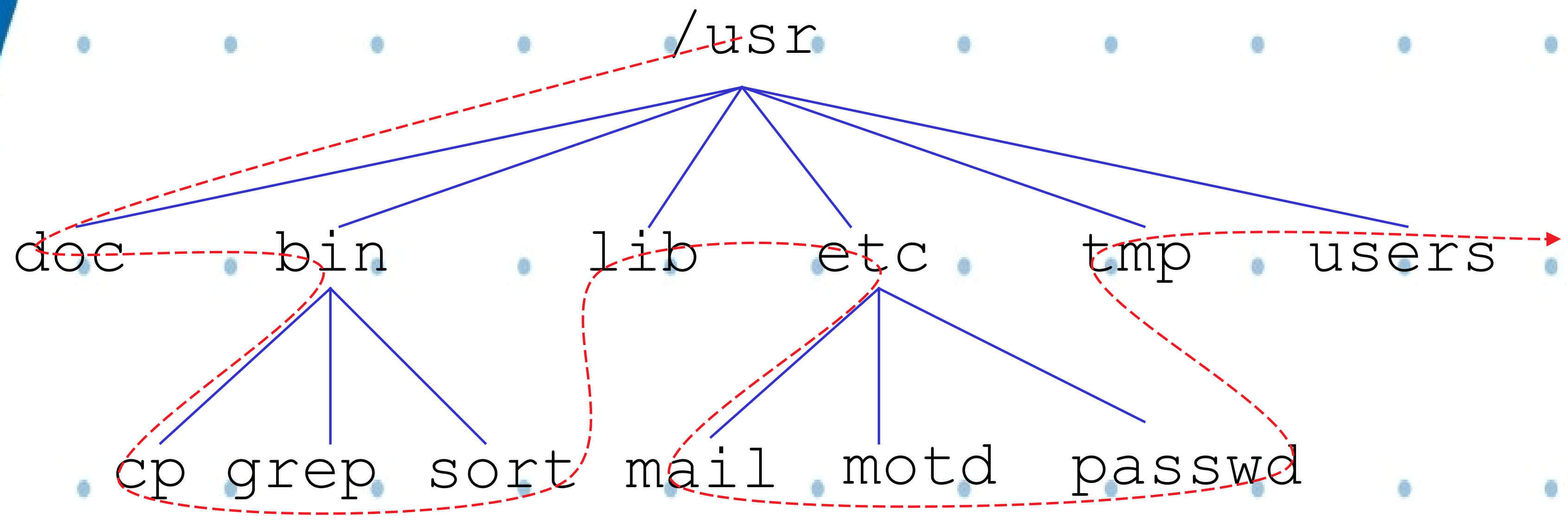




Recorridos

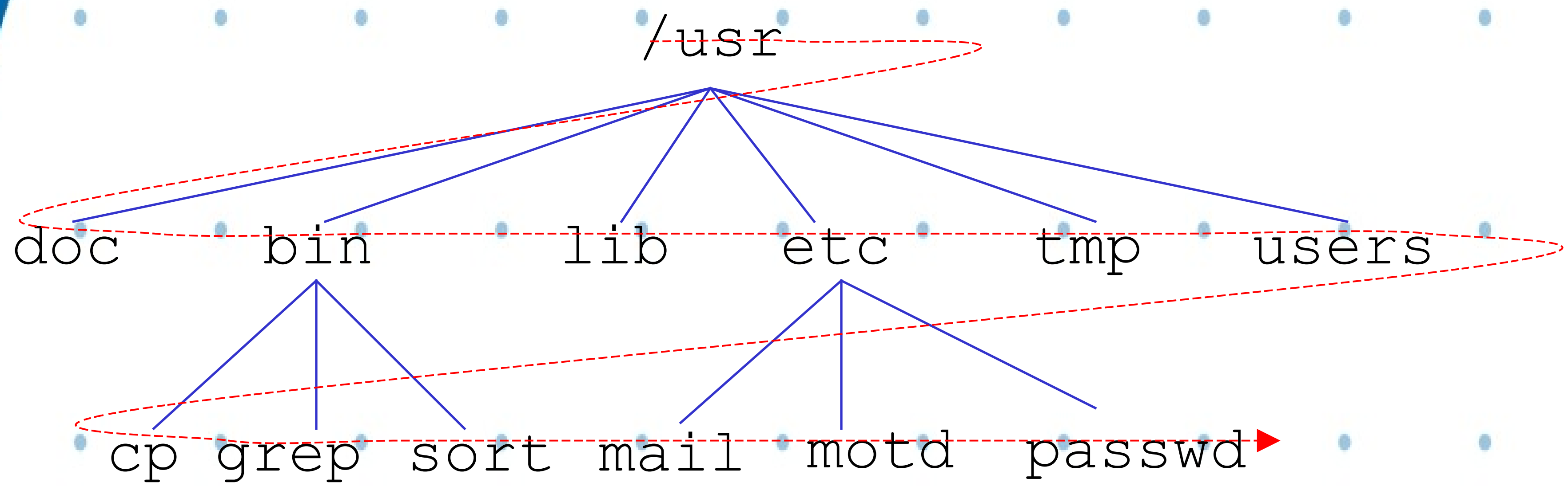
- **Preorden**
  Se procesa primero la raíz y luego los hijos
- **Inorden**
  Se procesa el primer hijo, luego la raíz y por último los restantes hijos
- **Postorden**
  Se procesan primero los hijos y luego la raíz
- **Por niveles**
  Se procesan los nodos teniendo en cuenta sus niveles, primero la raíz, luego los hijos, los hijos de éstos, etc.

Recorrido: Preorden



Ejemplo: Listado del contenido de un directorio

Recorrido: Por niveles



Recorrido: Preorden

```
public void preOrden() {
     imprimir (dato);
     obtener lista de hijos;
     mientras (tenga hijos) {
          hijo ⟵ obtenerHijo;
          hijo.preOrden();
     }
}
```

Recorrido: Por niveles

```
public void  porNiveles() {
      encolar(raíz);
      mientras cola no se vacíe {
          v ← desencolar();
          imprimir (dato de v);
          para cada hijo de v
          encolar(hijo);
      }
}
```